Towards Statically Checking Adherence to API Protocols

Jochen Quante
Robert Bosch GmbH, Corporate Research
Renningen, Germany
jochen.quante@de.bosch.com

Sushmita Suresh Naragund
TU Kaiserslautern
Kaiserslautern, Germany

Abstract

API protocols specify sequence constraints on API calls. They are typically available in form of finite state machines. Traditionally, API protocols are checked during runtime only: With each API call, the state in the state machine is tracked. If this leads to an error state (or an unsupported operation in a given state), the protocol is violated. However, it would be much more desirable to check adherence to the protocol statically, i.e., prior to execution of the code. In this paper, we report on our endeavors and experiences on doing such checks statically.

1 API Protocols

A given API typically comes with some assumptions on how it will be used. For example, consider a file I/O API. A file has to be opened before it can be read, and no more data can be read from it once it has been closed. A typical API protocol for file operations thus may look like this:

```
(fopen (fread|fwrite|fseek)* fclose)*
```

This regular expression specifies all allowed sequences of operations on this API. We treat API functions as the basic symbols, so valid function call sequences are the words of that language. Like any regular expression, this can be transformed to a deterministic finite automaton (DFA).

Of course, not all API protocols are expressible in a regular language. For example, it is not possible to specify that in a stack component, `pop` may only be called as many times as `push` has been called before: That would require a context-free language. We still stick to regular expressions due to practicability reasons; it is also the standard in related work on the topic [3]. For the stack example, we can still come up with a useful API protocol: It can ensure that we only `pop` when we know that there is something on the stack.

```
(push | isEmpty | isEmpty pop | push pop )*
```

2 Checking Protocol Adherence

We now want to check whether a given application that uses the API adheres to the protocol, i.e., if it obeys its sequence restrictions. Let us assume that we

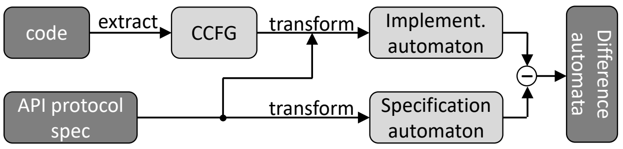

Figure 1: Approach

have another automaton that represents all possible API call sequences, which has been extracted from the code. Then our task can be rephrased as comparing the languages represented by the two automata.

Let us denote the language that a DFA A accepts by $L(A)$. The difference of the languages of two DFAs A and B is computed as $L(A)-L(B) = L(A \cap \overline{B})$. The complement of a DFA is derived by making accepting states non-accepting and vice versa. The intersection of two automata is computed by product construction. With these standard ingredients, we can calculate the language differences between protocol automaton P and implementation automaton I. They can be interpreted in the following way:

$L(I) \subseteq L(P)$	The API is correctly used.
$L(I) \not\subseteq L(P)$	The code violates the protocol.
$L(P) \subseteq L(I)$	The code uses the entire protocol.
$L(P) \not\subseteq L(I)$	The code does not use everything that is allowed by the protocol.

Furthermore, the corresponding difference automata (when non-empty) can provide hints about what exactly the violation or unused feature is. This can help a developer in quickly finding and fixing the underlying problem.

3 Simple Implementation Automaton

For a first evaluation of this idea, we implemented a simple static implementation automaton extraction approach and the automaton transformations as described above. It is sketched in Fig. 1 and works like this:

First, the control flow graph for each function is taken on basic block level. Each basic block is represented as a sequence of two nodes (entry and exit), and these nodes are connected according to the control flow between blocks. For basic blocks that con-

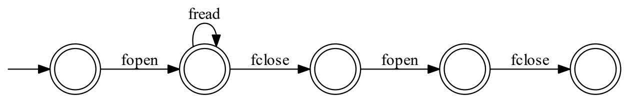

Figure 2: Initial extracted automaton.

tain calls, this sequence (of two nodes) is then transformed to a sequence of call and return edges. After this has been done for all functions, inlining of functions is performed: Each pair of call/return edges is supplemented by the graph of the respective function (if it exists, and if it is not a recursive call). Finally, unnecessary intermediate nodes and edges (subgraphs that contain no calls at all) are removed. The resulting graph is a combined call and control flow graph (CCFG) for a given function. Note that this graph may contain infeasible paths – it is an over-approximation of all possible paths through all function calls.

To convert this graph to an automaton, we create a non-deterministic automaton where each node is a state and each edge is a transition. Edges that correspond to a function call of the API under consideration become transitions labeled with the respective symbol, all other edges become ε edges. Only the exit node becomes an accepting state. Then, the automaton is converted to a deterministic one using subset construction. The result is an automaton that contains all API call sequences that can potentially occur in the code, along with infeasible ones.

Two problems occur for this very simple approach: Firstly, it does not distinguish between different instances of API usages, e.g., when different files are processed in parallel. Secondly, it abstracts away a lot of information so that the results are quite imprecise. However, the general transformation from CCFG to automaton can also be applied on the results of more advanced extraction algorithms. For example, object process graphs could be used instead [1] to increase precision.

4 Example

To evaluate this approach, we took the file API protocol from above and used a small compiler written in C as subject system. The compiler reads an input source file and ultimately writes an output binary file. We extracted the CCFG for the compiler code and translated it to an FSM. The result is shown in Fig. 2. Apparently, the sequence of operations can stop in any step. A quick check in the code reveals that this is due to error handling. So we add error handling to our protocol: After any operation, we may jump to an error final state. We choose `log_error` as the function that indicates this, so we enforce error logging in every error case. The resulting code automaton consists of 14 states, and the resulting difference automaton $I - P$ is partly shown in Figure 3. The difference

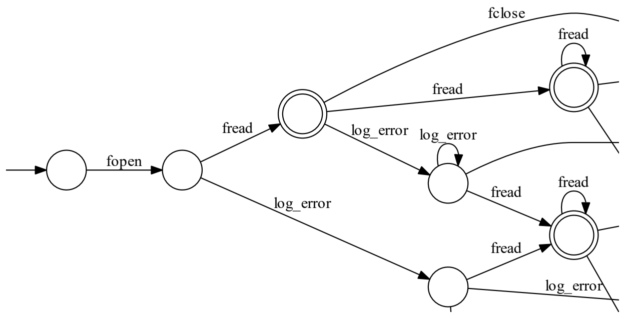

Figure 3: Difference automaton (implementation automaton minus protocol automaton) shows violations.

automaton is non-empty, which means the extended protocol is violated, and the language of the automaton consists of all violations. We can thus easily see from the Figure that, e.g., `fread` may be called after a `log_error`, and that the program may terminate after `fread` without logging. However, all that is subject to the limitation that these may be infeasible paths (in fact, most of them are). Furthermore, it turns out that `log_error` is too general to be used in the file I/O protocol – it is used in many other error cases as well.

5 Protocol Recovery

The technique introduced above can also be used to interactively reconstruct API protocols from code, in a way similar to the well-known reflexion analysis [2]: The user formulates an API protocol hypothesis, checks it against the code, gets feedback in the form of difference automata, adjusts the hypothesis, and iterates until the result is satisfactory.

6 Conclusion

We introduced a lightweight technique for checking API protocols against code. The technique can also be used to infer such protocols iteratively and thus support in program understanding. Despite the imprecise static extraction technique used, the results still give valuable insights into the analyzed program. We therefore conclude that the approach has the potential to be useful in practice.

References

[1] T. Eisenbarth, R. Koschke, and G. Vogel. Static object trace extraction for programs with pointers. *Journal of Systems and Software*, 77(3):263–284, 2005.

[2] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proc. of 3rd Symp. on Foundations of Software Engineering*, pages 18–28, 1995.

[3] M. Pradel, P. Bichsel, and T. R. Gross. A framework for the evaluation of specification miners based on finite state machines. In *Proc. of Int'l Conf. on Software Maintenance*, pages 1–10, 2010.